Distributed Task Scheduler

Suresh Podeti

7 min read · Sep 28, 2023

Photo by Olga Drach on Unsplash

Introduction

A **task** is a piece of computational work that requires resources (CPU time, memory, storage, network bandwidth, and so on) for some specified time.

A system that mediates between **tasks** and **resources** by intelligently allocating resources to tasks so that task-level and system-level goals are met is called a **task scheduler.**

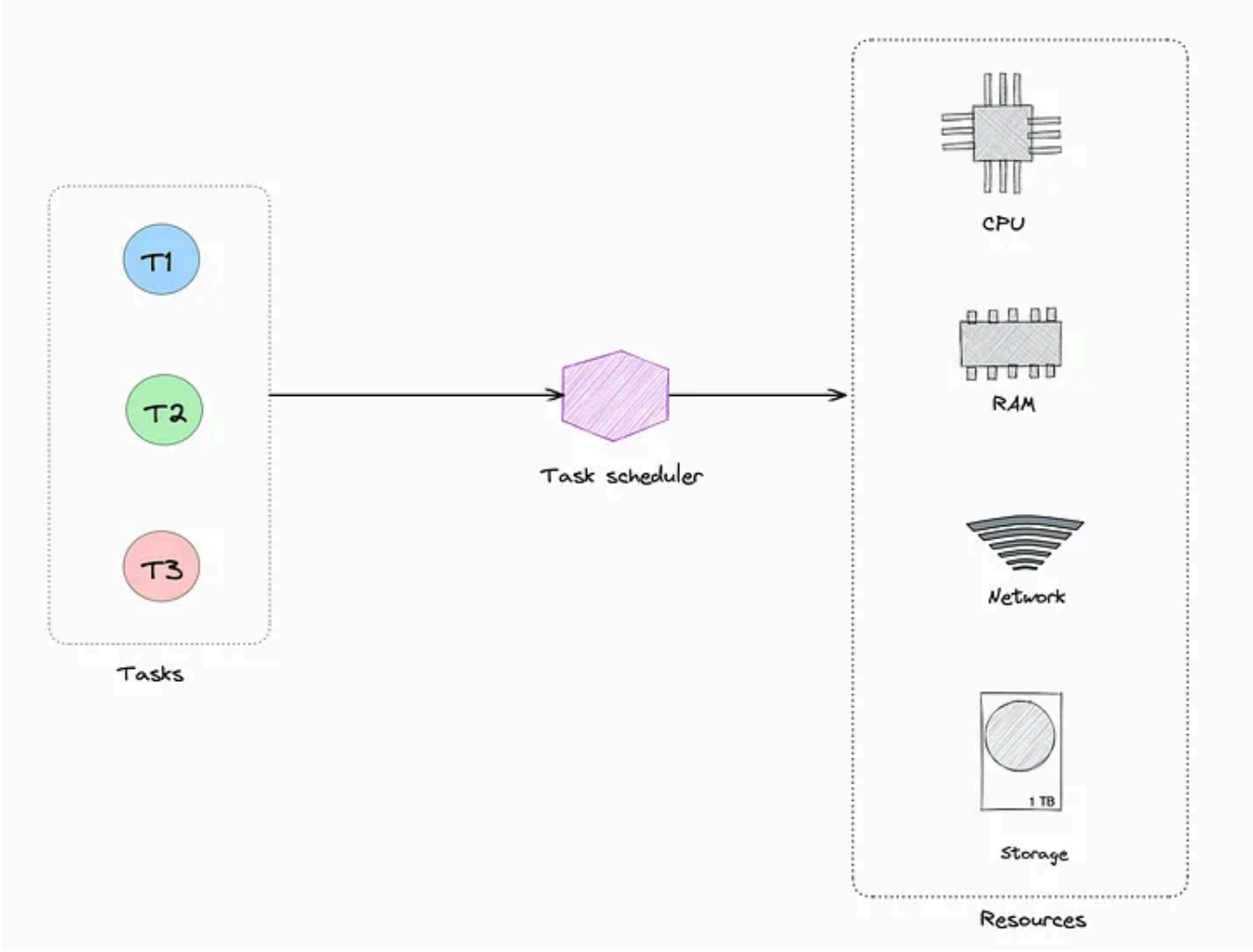

Fig 1.0: Task scheduler

The process of deciding and assigning resources to the tasks in a timely manner is called **task scheduling.**

Example is when we post a comment on Facebook. We don't hold the comment poster until that comment is delivered to all the followers. That delivery is **delegated to an asynchronous task scheduler** to do offline.

In a distributed system, many tasks run in the background against a single request by a user. Consider that there are millions to billions of users of a popular system like Facebook, WhatsApp, or Instagram. These systems require a task scheduler to handle billions of tasks. Facebook schedules its tasks against billions of parallel asynchronous requests by its users using Async.

Async is Facebook's own distributed task scheduler that schedules all its tasks. **Some tasks are more time-sensitive**, like the tasks that should run to notify the users that the livestream of an event has started. It would be pointless if the users received a notification about the livestream after it had finished. **Some tasks can be delayed**,

like tasks that make friend suggestions to users. Async schedules tasks based on appropriate priorities.

Requirements

- **Availability:** The system should be highly available to schedule and execute tasks.
- **Durability:** The tasks received by the system should be durable and should not be lost.
- **Scalability:** The system should be able to schedule and execute an ever-increasing number of tasks per day.
- **Bounded waiting time:** This is **how long** a task needs to **wait before starting execution**. We must not execute tasks much later than expected. Users shouldn't be kept on waiting for an infinite time. If the waiting time for users crosses a certain threshold, they should be notified.

Design

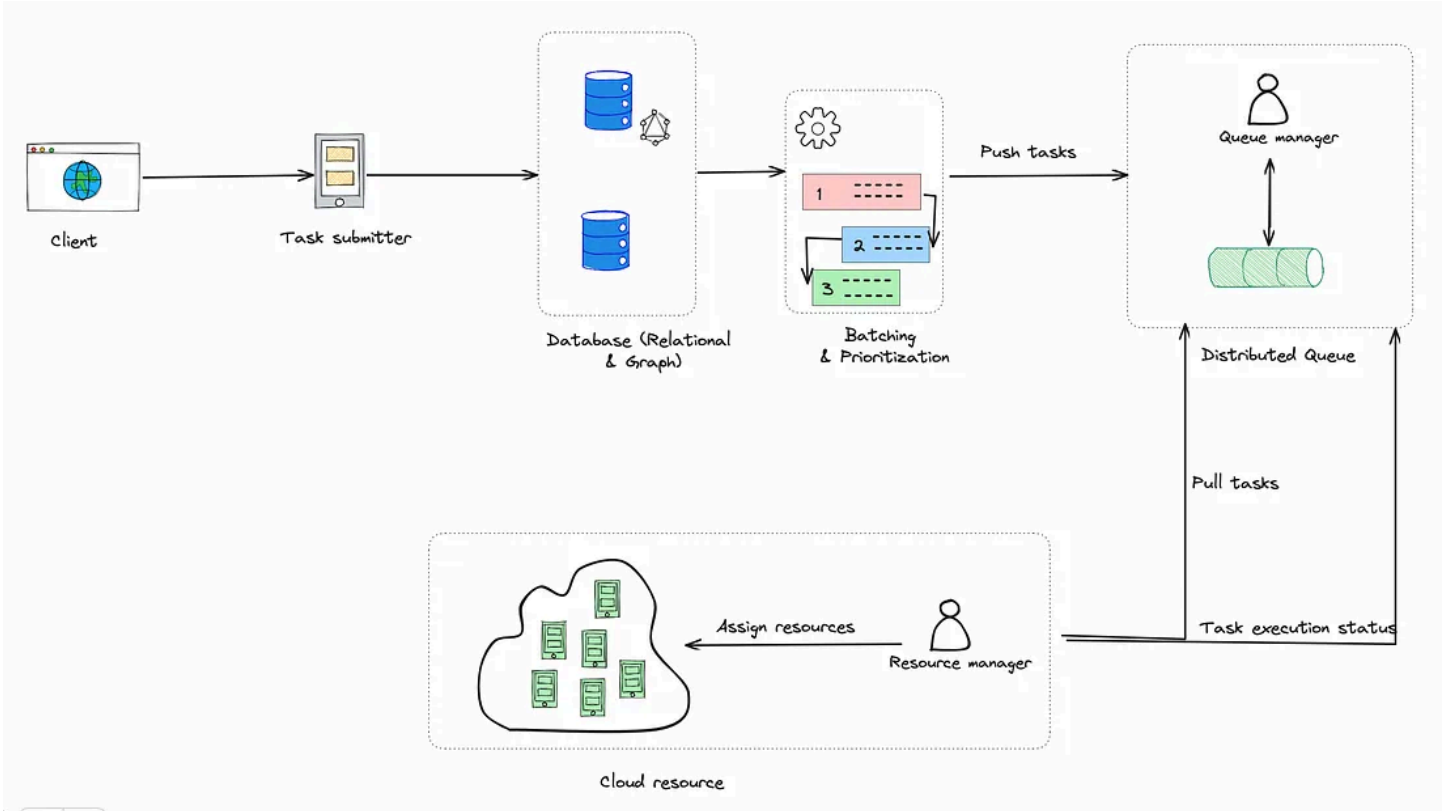

Fig 2.0: The design of task scheduler

Components

- **Task submitter**: The task submitter admits the task. There isn't a single task submitter. Instead, we have a cluster of nodes that admit the increasing number of tasks.

- **Database**: All of the tasks taken by the task submitter are stored in a distributed database. We use **relational database** to store — task IDs, user IDs, required resources, execution caps, the total number of attempts made by the client, delay tolerance, and so on. We use **Non-relational database** that uses the graph data structure to store data in a directed acyclic graph (DAG) of dependent tasks.

- **Batching and prioritization**: After we store the tasks in the RDB, the tasks are **grouped into batches**. Prioritization is based on the attributes of the tasks, such as delay tolerance or the tasks with short execution cap, and so on. The **top *K* priority tasks** are pushed into the distributed queue, where *K* limits the number of elements we can push into the queue. The value of *K* depends on many factors, such as currently available resources, the client or task priority, and subscription level.

- **Queue manager**: The queue manager adds, updates, or deletes tasks in the queue. It keeps track of the types of queues we use. It is also responsible for keeping the task in the queue until it executes successfully. In case a task execution fails, that task is made visible in the queue again. The queue manager knows which queue to run during the peak time and which queue to run during the off-peak time.

- **Resource manager**: The resource manager knows which of the resources are free. It pulls the tasks from the distributed queue and assigns them resources. The resource manager keeps track of the execution of each task and sends back their statuses to the queue manager. If a task goes beyond its promised or required resource use, that task will be terminated, and the status is sent back to the task submitter, which will notify the client about the termination of the task through an error message.

Execution cap

We have the following three categories for our tasks:

- Tasks that can't be delayed — Urgent tasks.

- Tasks that can be delayed.

- Tasks that need to be executed **periodically** (for example, every 5 minutes, or every hour, or every day) — Periodic tasks.

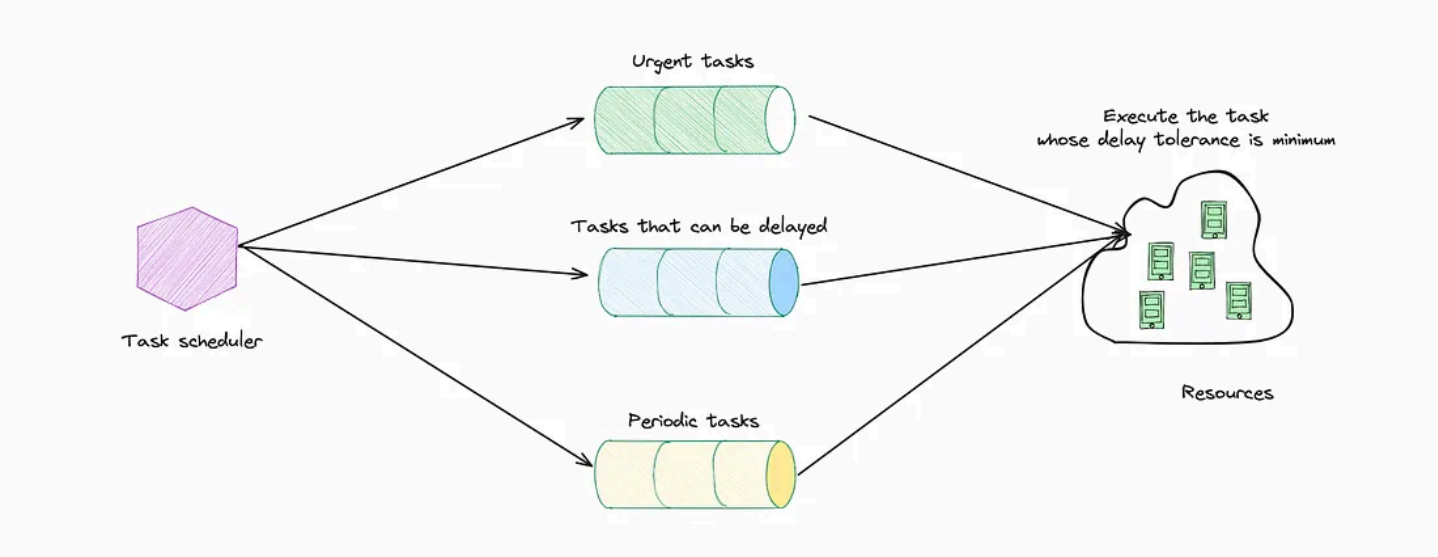

Fig 3.0: Multiple queues based on the task categories

Our system ensures that tasks in non-urgent queues are not starved. As soon as some task's delay limit is about to be reached, it is moved to the urgent tasks queue so that it gets service.

Prioritization

Some tasks take very long to execute and occupy the resource blocking other tasks. The **execution cap** is an important parameter to consider while scheduling tasks. If we completely allocate a resource to a single task and wait for that task's completion, some tasks might not halt because of a bug in the task script that doesn't let it finish its execution. We let the clients set the **execution cap for their tasks**. After that specified time, we should stop task execution, release the resource, and allocate it to the next task in the queue. If the task execution stops due to the execution cap limit, our system notifies the respective clients of these instances. The client needs to do appropriate remedial actions for such cases.

Task idempotency

There are tasks that need urgent execution. For example, in a social application like Facebook, the users can mark themselves safe during an emergency situation, such as an earthquake. The tasks that carry out this activity should be executed in a timely manner, otherwise this feature would be useless to Facebook users. Sending an email notification to the customers that their account was debited a certain amount of money is another example of tasks that require urgent execution.

To prioritize the tasks, the task scheduler maintains a **delay tolerance** parameter for each task and executes the task close to its delay tolerance. **Delay tolerance** is the maximum amount of time a task execution could be delayed. The task that has the shortest delay tolerance time is executed first. By using a delay tolerance parameter, we can postpone the tasks with longer delay tolerance values to make room for urgent tasks during peak times.

Resource capacity optimization

There could be a time when resources are close to the overload threshold (for example, above 80% utilization). This is called **peak time**. The same resource may be idle during off-peak times. So, we have to think about better utilization of the resources during off-peak times and how to keep resources available during peak times.

There are tasks that **don't need urgent execution**. For example, in a social application like Facebook, suggesting friends is not an urgent task. We can make a separate queue for tasks like this and **execute them in off-peak times**. If we consistently have more work to do than the available resources, we might have a capacity problem, and to solve that, we should commission more resources.

Task idempotency

If the task executes successfully, but for some reason the machine fails to send an acknowledgement, the scheduler will schedule the task again. The task is executed again.

We don't want the final result to change when executing the task again. This is critical in financial applications while transferring money. We require that tasks are idempotent. An idempotent task produces the same result, no matter how many times we execute it.

This property is added in the implementation by the developers where they identify the property by something (for example, its name) and overwrite the old one.

Evaluation

Availability

Task submission is done by several nodes. If a node that submits a task fails, the other nodes take its place. The queue in which we push the task is also distributed in nature, ensuring availability. We always have resources available because we

continuously monitor if we need to add or remove resources. Each component in the design is distributed and makes the overall system available.

Durability

We store the tasks in a persistent distributed database and push the tasks into the queue near their execution time. Once a task is submitted, it is in the database until its execution.

Scalability

Our task scheduler provides scalability because the task submitter is distributed in our design. We can add more nodes to the cluster to submit an increasing number of tasks. The tasks are then saved into a distributed relational database, which is also scalable. The tasks from RDB are then pushed to a distributed queue, which can scale with an increasing number of tasks. We can add more queues for different types of tasks. We can also add more resources depending on the resource-to-demand ratio.

Fault tolerance

A task is not removed from the queue the first time it is sent for execution. If the execution fails, we retry for the maximum number of allowed attempts. If the task contains an infinite loop, we kill the task after some specified time and notify the user.

Task Scheduler | Software Architecture | Software Architect | Distributed Systems | System Design Interview

Edit profile

Written by Suresh Podeti

1.2K Followers